

# Towards a SDL-DEVS Simulator

## Multiparadigm simulation

Pau Fonseca i Casas

Statistics and Operations Research Department  
Universitat Politècnica de Catalunya  
Barcelona, Catalunya, Spain  
pau@fib.upc.edu

Josep Casanovas

Statistics and Operations Research Department  
Universitat Politècnica de Catalunya  
Barcelona, Catalunya, Spain  
josepk@fib.upc.edu

**Abstract**— In this paper, we present the first version of a simulator that allows executing models defined using Discrete Event System Specification and models defined using Specification and Description Language. Specification and Description Language (SDL) is a graphical language, standardized under the ITU Z.100 recommendation, widely used to represent telecommunication systems, process control and real-time applications in general. Discrete Event System Specification (DEVS) is a formalism widely used on the simulation field to represent Discrete Event Systems. The execution of the DEVS models is based on a transformation of the simulation model DEVS representation to an equivalent SDL representation. To do this, we propose a XML representation for the DEVS models, and a XML representation for SDL models. Also we implement an algorithm capable to perform this transformation.

**Keywords**—simulation; formal language; SDL; DEVS

### I. SPECIFICATION AND DESCRIPTION LANGUAGE

Specification and Description Language (SDL) is an object-oriented, formal language defined by the International Telecommunication Union – Telecommunication Standardization Sector (ITU-T). The recommendation that summarizes its use is Z.100. The language is designed to specify complex, event-driven, real-time, interactive applications involving many concurrent activities using discrete signals to enable communication [1].

The definition of the model is based on different components:

- Structure: system, blocks, processes and processes hierarchy.
- Communication: signals, with the parameters and channels that the signals use to travel.
- Behavior: defined through the different processes and procedures.
- Data: based on Abstract Data Types (ADT).
- Inheritances: to describe the relationships between, and specialization of, the model elements.

The language has 4 levels (Figure 1), (i) System, (ii) Blocks, (iii) Processes and (iv) Procedures. To know more about the

Specification and Description Language please refers to [2] [3] or Z.100 recommendation [1].

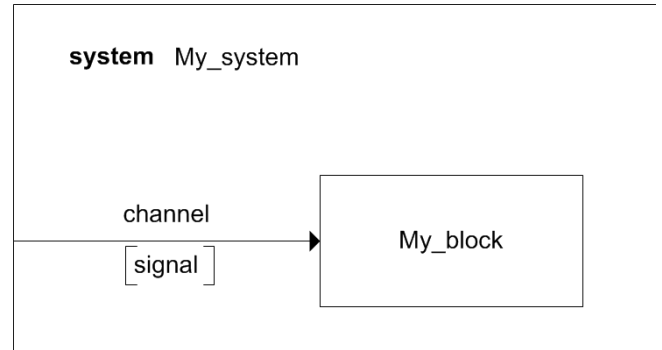


Figure 1: The first level of an SDL model.

### II. GRAPHICAL AND NO GRAPHICAL LANGUAGE

SDL have two representations, SDL PR and SDL GR. SDL-PR is conceived to be easily processed by computers, also allows a compact representation of a model, while SDL-GR has some textual elements which are identical to SDL-PR (this is to allow specification of data and signals) it is mainly graphical. Figure 2 shows an example of a textual and graphical representation of an SDL process.

We are not using the textual version of SDL only for one reason. Some different textual representations of DEVS based on XML format exist. Since we want to allow an automatic transformation from SDL to DEVS, the use of XML simplifies our programming code because now is easy to read and write structured text files that follow the XML syntax, and also, thanks to the XSD we can validate the correctness of its syntax. We are using the XML representation for SDL proposed in [4]. Since the more important aspects of an XML file can be represented, and validated, through an XSD file, in the next section some areas of the XSD file are shown.

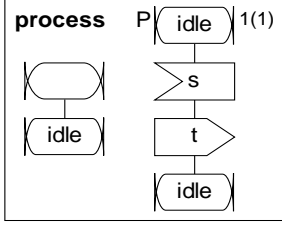


Figure 2: textual and graphical SDL representation

#### A. XML representation of an SDL simulation model

This representation was first presented on [5], no modifications have been done from this schema. We next describe the more important elements. For further details, please see [5], or download the complete schema from [6].

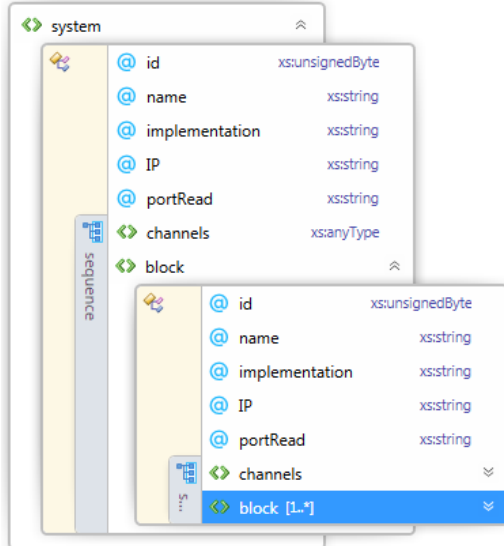


Figure 3. XSD schema, system view

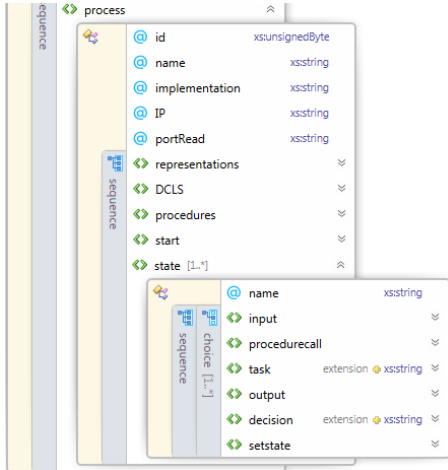


Figure 4. XSD schema, process view

In Figure 3 we show the first level of the XSD schema we use to validate the structure of our XML. The first level of

this schema represents the first level of the Specification and Description Language (system outmost block). Figure 4 shows the process type that allows represent an SDL process.

### III. DEVS FORMALISM

Proposed by Bernard Zeigler in the 70's, the main scope of Discrete Event System Specification (DEVS) is the representation of simulation models. A DEVS model is a tuple composed by the elements defined as follows:

$M = \langle Z, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta \rangle$  **where,**

$X$ : set of input values

$S$ : set of state values

$Y$ : set of output values

$\delta_{int}$ : internal transition function;  $\delta_{int}: S \rightarrow S$

$\delta_{ext}$ : external transition function;  $\delta_{ext} \rightarrow Q \times X \rightarrow S$

$Q = \{(s, e) | s \in S, 0 \leq e \leq ta(s)\}$ : set of states

$e$ : time from the last transition

$\lambda: S \rightarrow Y$ : output function

$ta$ : time advance function

$ta: S \rightarrow R_0^+$

DEVS distinguish between an internal and external transition. An internal transition is a kind of transition that doesn't need any external event to be launched. As an example, if in a "t" time, the system reach the state "s", the system remains in this state the during the time defined on a "time advance" function "ta(s)" (if no external event is received). When the time reach the value defined in the "ta(s)" function an output event is produced (this output is defined on the " $\lambda(s)$ " function) and the state changes to "s' ". This process is defined in the internal transition  $s' = \delta_{int}(s)$ .

External transitions define the modifications in the model due to the reception of external events. For example, before the model reach the state "s' ", in a time "t", due to his internal transition, an external event, with value x, is processed. In this case the system reach state (s,e) where  $e < ta(s)$ , the transition follows the external transition function, defined by  $s' = \delta_{ext}(s, e, x)$ , and no exit event is produced.

At this point it is important to underline that "ta(s)" could be any real number, plus 0 and  $\infty$ , and:

- If ta(s) is 0, "s" is a *transitory* state.
- If ta(s) =  $\infty$ , "s" is a *passive* state.

In the next lines we review two examples from [7]. We use these two models to transform them automatically to a SDL specification and then execute the models using SDLPS [4].

#### A. Processor example

This example represents a single processor that receives different jobs. Each job has associated a processing time (represented by a real number). Once the time is over event "ready" is produced. When a new event reach the processor, if this is working with a job, this event is ignored.

The DEVS formalization of this model is:

$$\begin{aligned}
M = & \langle X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta \rangle \textbf{ where}, \\
X = & \{job_1, job_2, \dots, job_n\} \\
S = & \{job_1, job_2, \dots, job_n\} \cup [\emptyset] \times R^+ \\
Y = & \{y(job_1), y(job_2), \dots, y(job_n)\} \\
\delta_{int}(job, \sigma) = & (\emptyset, \infty) \\
\delta_{ext}(job, \sigma, e, x) = & \begin{cases} (x, tp(x)) & \text{if } job = \emptyset \\ (job, \sigma - e) & \text{otherwise} \end{cases} \\
\lambda(job, \sigma) = & y(job) \\
ta(job, \sigma) = & \sigma
\end{aligned}$$

#### B. FIFO Queue example

The queue represented in this example has the following characteristics:

- The queue has infinite capacity.
- Different jobs reach the queue to be stored, while the “ready” signals symbolize the necessity of transmit the first job of the queue.
- The transmission of this job is done through an output event.
- The queue spends 0 time units in the exit delay.

The DEVS model is:

$$\begin{aligned}
M = & \langle X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta \rangle \textbf{ where}, \\
X = & \{job_1, job_2, \dots, job_n\} \cup \{ready\} \\
S = & \{job_1, job_2, \dots, job_n\} \cup [\emptyset] \times R^+ \\
Y = & \{y(job_1), y(job_2), \dots, y(job_n)\} \\
\delta_{int}(q \cdot job, \sigma) = & (q, \infty) \\
\delta_{ext}(job, \sigma, e, x) = & \begin{cases} (x \cdot q, \infty) & \text{if } x \in J \\ (q, 0) & \text{otherwise} \end{cases} \\
\lambda(q \cdot job, \sigma) = & job \\
ta(job, \sigma) = & \sigma
\end{aligned}$$

#### IV. DEVS COUPLED MODELS

DEVS also allows formalize simulation models without describing the behavior for each element belonging to the model. It is possible to describe the structural relations that exist among identical elements. These models are named “coupled models”. In DEVS there are two main types of coupled models:

- Modular coupling.
- Non modular coupling.

In modular coupling integration among different model components happens only across entries and exits defined in the components, while in non-modular coupling, interaction is produced across states. The literature established that it is possible to pass from one kind of coupling model to the other [5], therefore in present paper we will focus on show the existing relation among SDL formalism and the DEVS modular formalism.

For simplicity the DEVS coupled model used in this paper is DEVS coupled model with ports. In this model a

series of input and output ports are described. With this logic is possible to depict the following example, see Figure 5, representing the combination of the two models that have been defined previously (the queue and the processor).

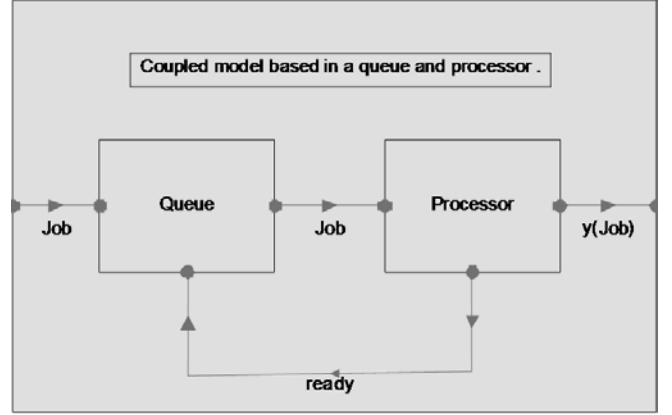


Figure 5. DEVS coupled model.

The coupling model specification for this model is:

$$\begin{aligned}
N = & (X, Y, D, \{Md \mid d \in D\}, EIC, EOC, IC, Select), \text{ on} \\
X = & J \times \{\text{inport1}\} \\
Y = & \{y(Job) \mid Job \in J\} \times \{\text{outport1}\} \\
D = & \{P, Q\} \\
EIC = & \{(N, \text{inport1}), (Q, \text{inport1})\} \\
EOC = & \{(P, \text{outport1}), (N, \text{outport1})\} \\
IC = & \{(P, \text{outport2}), (Q, \text{outport2})\}
\end{aligned}$$

#### V. XML REPRESENTATION OF DEVS MODELS

Some attempts have been made to represent DEVS models using XML. As an example [8] presents a schema that cannot characterize the programming logic, loops and if-then-else constructs. Our approach is going further and allows the representation of those elements. We propose to use ANSI C (since it is an ISO standard) to represent the code contained in model. Also this simplifies the representation of the model on SDL, using a variant named SDL-RT who uses ANSI C too. In our point of view the DEVS-XML representation that we present here can be considered as a good starting point for a robust and complete representation of DEVS models using XML.

We follow some conventions to represent a DEVS model using XML syntax:

All the code needed to fully define the simulation model is defined on the “values” xml section.

The initial conditions of the model is defined in the XML as well, using a “value” attribute related to all the variables that defines the state of an atomic DEVS model.

Also, to represent the value  $\infty$  used in the passive states we use ‘inf’ literal value.

Some parts of the XML schema used to represent coupled and an atomic models is shown in Figure 6.

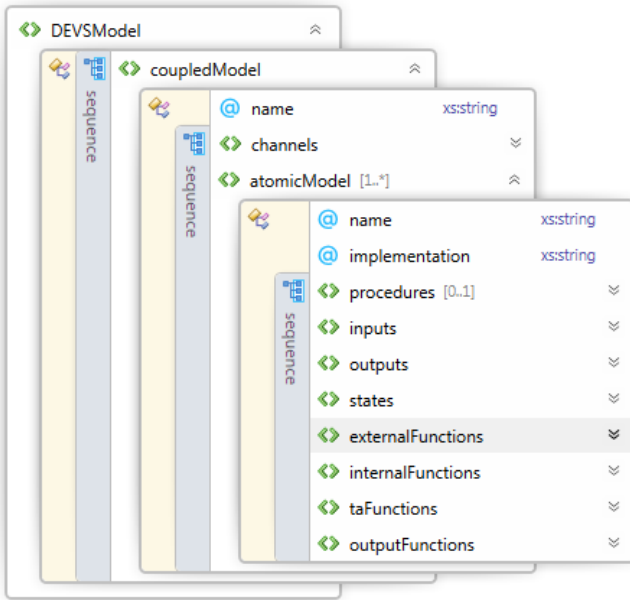


Figure 6. DEVS XML schema

The complete definition of the *DEVModel* using XML is shown next. On Figure 7 is represented the whole DEVS model using XML. On Figure 8 the definition of the *states* is shown. Figure 9 shows the definition of the *input* and the *output* elements. Figure 10 represents the *external functions* and in Figure 11 the *time advance* and *output functions*.

From this DEVS-XML representation we can obtain an equivalent model described using Specification and Description Language, using again XML (SDL-XML).

```
<?xml version="1.0" encoding="UTF-8"?>
<DEVModel xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:
  <coupledModel name="GG1">
    <channels>
      <channel name="in" start="queue" end="processor1" dual="no">
        <event name="job_id"></event>
      </channel>
      <channel name="out" start="processor1" end="queue" dual="no">
        <event name="job_id"></event>
      </channel>
    </channels>
    <atomicModel name="queue" implementation="CDEVQueue">
      <procedures>...</procedures>
      <inputs>...</inputs>
      <outputs>...</outputs>
      <stateVariables>...</stateVariables>
      <internalFunctions>...</internalFunctions>
      <externalFunctions>...</externalFunctions>
      <taFunctions>...</taFunctions>
      <outputFunctions>...</outputFunctions>
    </atomicModel>
    <atomicModel name="processor1">
      <procedures>...</procedures>
      <inputs>...</inputs>
      <outputs>...</outputs>
      <states>...</states>
      <externalFunctions>...</externalFunctions>
      <internalFunctions>...</internalFunctions>
      <taFunctions>...</taFunctions>
      <outputFunctions>...</outputFunctions>
    </atomicModel>
  </coupledModel>
</DEVModel>
```

Figure 7. GG1 DEVS model.

```
<atomicModel name="processor1">
  <procedures>...</procedures>
  <inputs>...</inputs>
  <outputs>...</outputs>
  <states>
    <variables>
      <!--
        The initial conditions of the model is defined on the value.
      -->
      <variable name="job" type="Integer" value="0"></variable>
      <variable name="processing_time" type="Real" value="'inf'"></variable>
      <!--
        In this case it is not needed a literal variable to define the states.
        This is an example of how discrete states can be defined on a model.
      -->
      <variable name="state" type="literal" value="idle|working"></variable>
    </variables>
  </states>
  <externalFunctions>...</externalFunctions>
  <internalFunctions>...</internalFunctions>
  <taFunctions>...</taFunctions>
  <outputFunctions>...</outputFunctions>
</atomicModel>
```

Figure 8. States definition.

```
<atomicModel name="processor1">
  <procedures>...</procedures>
  <inputs>
    <port name="in">
      <signal name="job_id" type="Integer"></signal>
    </port>
  </inputs>
  <outputs>
    <port name="out">
      <signal name="job_id" type="Integer"></signal>
    </port>
  </outputs>
  <states>...</states>
  <externalFunctions>...</externalFunctions>
  <internalFunctions>...</internalFunctions>
  <taFunctions>...</taFunctions>
  <outputFunctions>...</outputFunctions>
</atomicModel>
```

Figure 9. Input and output elements.

```
<atomicModel name="processor1">
  <procedures>...</procedures>
  <inputs>...</inputs>
  <outputs>...</outputs>
  <states>...</states>
  <externalFunctions>
    <function port="in" job="Integer" processing_time="Real" x="">
      <condition>
        <code>job==0</code>
      </condition>
      <body>
        <setstate job="job_id" processing_time="tp(job_id)"></setstate>
      </body>
    </function>
  </externalFunctions>
  <internalFunctions>
    <function job="Integer" processing_time="Real">
      <condition>
        <code>true</code>
      </condition>
      <body>
        <setstate job="0" processing_time="'inf'"></setstate>
      </body>
    </function>
  </internalFunctions>
  <taFunctions>...</taFunctions>
  <outputFunctions>...</outputFunctions>
</atomicModel>
```

Figure 10. External and internal functions.

```

<atomicModel name="processor1">
  <procedures>...</procedures>
  <inputs>...</inputs>
  <outputs>...</outputs>
  <states>...</states>
  <externalFunctions>...</externalFunctions>
  <internalFunctions>...</internalFunctions>
  <taFunctions>
    <function job="Integer" processing_time="Real">
      <condition>
        <code></code>
        <body>
          <return value="processing_time"></return>
        </body>
      </condition>
    </function>
  </taFunctions>
  <outputFunctions>
    <function job="Integer" processing_time="Real">
      <condition>
        <code></code>
        <body>
          <send port="out" signal="job_id">job;</send>
        </body>
      </condition>
    </function>
  </outputFunctions>
</atomicModel>

```

Figure 11. Time advance and output functions.

## VI. TRANSFORMING FROM DEVS TO SDL

The transformation algorithm is based on the theoretical proposal presented on [9]. In this infrastructure we implement this proposal using the XML representation for the SDL and DEVS model (DEVS-XML and SDL-XML). This allows us to obtain a new XML file that represents a DEVS model. The schema used here to represent the SDL model is based on those presented on [5] we only show here the more important aspects of the resulting XML file that represents the new proposal for the DEVS-XML representation.

```

<?xml version="1.0"?>
<system id="0" name="GG1" implementation="" IP="" portRead="">
  <channels>...</channels>
  <process id="" name="queue" implementation="" IP="" portRead="">
    <DCLS>...</DCLS>
    <procedures>...</procedures>
    <start>...</start>
    <state name="joblist='Intege'>...</state>
  </process>
  <process id="" name="processor1" implementation="" IP="" portRead="">
    <DCLS>...</DCLS>
    <procedures>...</procedures>
    <start>...</start>
    <state name="job='Integer' p">...</state>
  </process>
</system>

```

Figure 12. XML representation of the SDL model.

On Figure 12 we can see the whole representation of the DEVS-XML model, now transformed to a SDL-XML representation. We can see, as we can expect, that the model contains two processes, the *queue* and the *processor1*.

```

<process id="" name="queue" implementation="" IP="" portRead="">
  <DCLS>...</DCLS>
  <procedures>...</procedures>
  <start>...</start>
  <state name="joblist='IntegerList' processing_time='Real' ">
    <input id="1" name="INT1"></input>
    <decision id="2" name="" iftrue="3" iffals="5">true</decision>
    <task id="3" name="">
      processing_time="inf";
      joblist=remove_first_list_element(joblist,job_id);
    </task>
    <output id="4" name="INT1" self="yes" to="" via="">
      <param name="delay" value="processing_time"></param>
      <param name="priority" value="0"></param>
    </output>
    <setstate id="5" name="joblist='IntegerList' processing_time='Real' "></setstate>
    <input id="6" name="EXT1"></input>
    <decision id="7" name="" iftrue="3" iffals="9">is_job(x)</decision>
    <task id="8" name="">
      joblist=add_new_job_to_list(x,joblist);
      processing_time="inf";
    </task>
    <setstate id="9" name="joblist='IntegerList' processing_time='Real' "></setstate>
    <input id="10" name="EXT1"></input>
    <decision id="11" name="" iftrue="12" iffals="13">true</decision>
    <task id="12" name="">
      processing_time="0";
      joblist=add_new_job_to_list(x,joblist);
      processing_time="inf";
    </task>
    <setstate id="13" name="joblist='IntegerList' processing_time='Real' "></setstate>
  </state>
</process>

```

Figure 13. Process queue definition.

On Figure 13 the XML representation using SDL for the DEVS queue element is shown.

## VII. SIMULATING THE DEVS MODEL ON SDLPS

Regarding the infrastructure used, it is remarkable that SDLPS has been build using C++ and C languages. The code related to the model is represented using a DLL, and the generation of the SDL-XML model is done through a plug-in on Microsoft Visio®.

On the next figure we can see the DEVS GG1 model on SDLPS. Note that it is not represented the DEVS model because the Microsoft Visio® plug-in we develop allows the generation of the SDL-XML from a SDL Microsoft Visio® diagram, but the inverse is not yet implemented (we cannot regenerate the diagram form a SDL-XML representation).

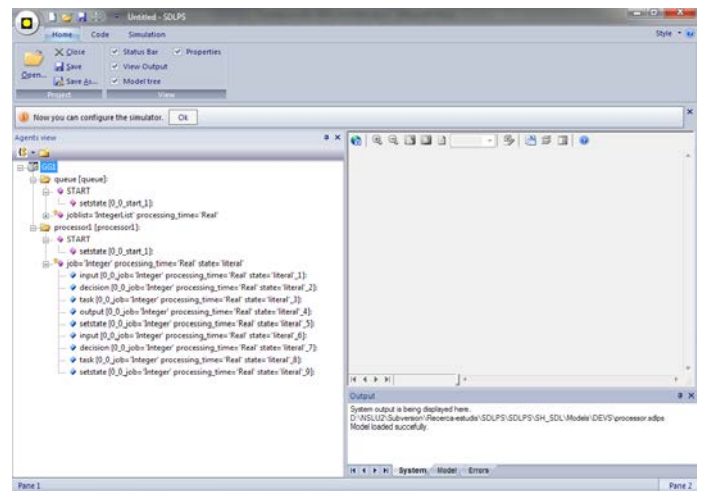


Figure 14. SDLPS system loading the DEVS model.

On the left side we can see the tree that contains all the elements that defines the model.

### VIII. DISCUSSION

Several formal languages exist that can be used to represent a simulation model, like SDL, DEVS, PetriNets [10], or SysML among others. The use of this kind of languages in a simulation project is very desirable, because clearly differentiates the model from the implementation that finally represents the model. Also helps in the understanding of the model and helps in the Validation and Verification process. However, only few simulators allow working with different formal languages in the same environment.

In this paper we present a XML representation for atomic and coupled DEVS models with the main goal to serve as a starting point to achieve a complete representation of a DEVS model. This allows the construction of tools that works with DEVS. Also we shown that thanks this representation we can implement a transformation algorithm between DEVS and SDL, allowing that in a single model we can use both formalisms. This simplifies the reuse of simulation models, and the collaboration between different groups that can use the formal language they prefer to define the models. The first issue that is needed to be fixed is that only few of them have been standardized, this often implies that the XML representation of the models needs to assure the inclusion of new features to the language. Also the textual representation of these models, needed in order to be used in a computer simulator, sometimes does not exist.

Also in this paper we present an infrastructure that allows the simulation of DEVS and SDL models. This combination of both languages can be done thanks the XML representation used for DEVS and SDL models. In this infrastructure we show that the final user can define the models using common simulation tools, like Microsoft Visio®, and thanks a plug-in the XML representation can be obtained.

Now this infrastructure is currently used in a production environment in real simulation projects for different well known industries. Those projects help us in the Verification of the tool and in the development of some missing plug-ins for some of the more common used computer programs in the industry.

### ACKNOWLEDGMENT

Many thanks to the Computing Laboratory of the Barcelona Informatics School of the Polytechnic University of Catalonia for his support in the development of this project.

### REFERENCES

- [1] Telecommunication standardization sector of ITU. (1999) Series Z: Languages and general software aspects for telecommunication systems. [Online]. <http://www.itu.int/ITU-T/studygroups/com17/languages/Z100.pdf> (accessed 15/09/2011)
- [2] Lauren Doldi, *Validation of Communications Systems with SDL: The Art of SDL Simulation and Reachability Analysis*.: John Wiley & Sons, Inc., 2003.
- [3] Rick Reed, "SDL-2000 form New Millenium Systems," *Elektronikk 4.2000*, pp. 20-35, 2000.
- [4] Pau Fonseca i Casas, "SDL distributed simulator," in *Winter Simulation Conference 2008*, Miami, 2008, <http://wintersim.org/abstracts08/POS.htm#fonsecaicasas> p84590.
- [5] Pau Fonseca i Casas, "Towards an automatic transformation from a DEVS to a SDL specification," in *Proceedings of the 2009 Summer Simulation Multiconference*, Istanbul, Turkey, 2009.
- [6] Pau Fonseca i Casas. (2011) Pau Fonseca i Casas. [Online]. <http://www-eio.upc.es/~pau/index.php?q=node/30> (accessed 15/09/2011)
- [7] b.p. Zeigler, h. Praehofer, and d. Kim, *Theory of Modeling and Simulation*.: Academic Press, 2000.
- [8] J.L. Risco-Martín, S. Mittal, M.A. López-Peña, and J.M. De la Cruz, "A W3C XML Schema for DEVS Scenarios," in *Spring Simulation Multiconference 2007*, vol. DEVS Symposium, Norfolk, Virginia, 2007, pp. 279-286.
- [9] Pau Fonseca i Casas and Josep Casanovas Garcia, "Using SDL diagrams in a DEVS specification," in *The Fifth IASTED International conference on Modeling Simulation and Optimization*, 2005.
- [10] L. Recalde, E. Teruel, and E. Silva, "Autonomous continuous P/T systems. Application and Theory of Petri Nets," *Lecture Notes in Computer Science*, pp. 107-126, 1999.